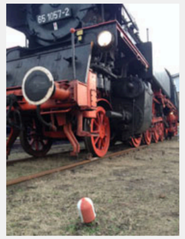

BACK
to the
ROOTS

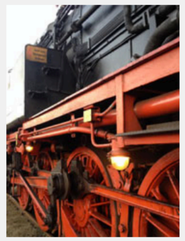
BEISPIEL: BILDERGALERIE

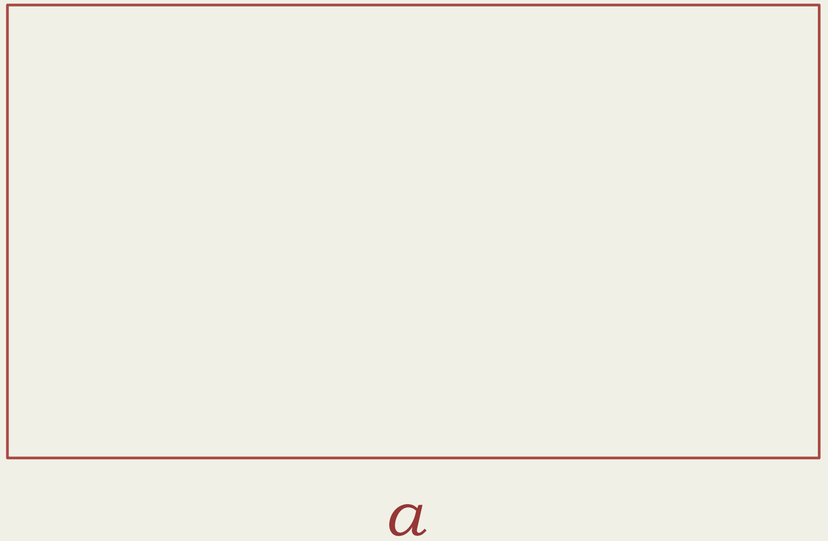









$$a \cdot b = A$$

$$a \,/\, b = r$$

$$a = r \cdot b$$

$$r \cdot b \cdot b = A$$

$$b^2 = A \,/\, r$$

$$b = \sqrt{A \,/\, r}$$

$$a = r \cdot \sqrt{A \,/\, r}$$

$$a = \sqrt{A \cdot r}$$

HERON-VERFAHREN (BABYLONISCHES WURZELZIEHEN)

$$x_0 \approx \sqrt{a} > 0$$

$$x_{n+1} = (x_n + a \,/\, x_n) \,/\, 2$$

$$\lim_{n \to \infty} x_n = \sqrt{a}$$

$a = 2$

$x_0 = 1$

$x_1 = (1 + 2 \,/\, 1) \,/\, 2 = 1.5$

$x_2 = (1.5 + 2 \,/\, 1.5) \,/\, 2 = 1.41666667$

$x_3 = (1.41666667 + 2 \,/\, 1.41666667) \,/\, 2 = 1.41421568...$

$\sqrt{2} = 1.41421356...$

Mathematische Funktionen mit Sass

Zurück zu den Wurzeln

Mit Sass könnt ihr mathematische Funktionen wie Wurzelziehen umsetzen. Wir zeigen euch anhand einer Bildergalerie, wozu das nützlich ist und wie ihr euch geschickt dem Wert für die Wurzel nähert.

Auf einer responsiven Webseite möchten wir eine responsive Bildergalerie einbauen. D.h. die Größen sollen nicht in em oder rem oder gar px angegeben sein, sondern die Bilder sollen ihre Größe dem Viewport anpassen. Die Bilder können unterschiedliche Seitenverhältnisse haben und im Quer- oder Hochformat vorliegen. Wir geben ihnen quadratische Rahmen, die an die Diasammlung unserer Eltern erinnen. Als Markup verwenden wir:

```
<ul class="gallery">
  <li>
    <div>
      <img src="…" alt="…"/>
    </div>
  </li>
  <li>
    <div>
      <img src="…" alt="…"/>
    </div>
  </li>
  ⋮
</ul>
```

Ihr werdet fragen: Wozu die div? Dazu gleich mehr.

Der Autor

Gunnar Bittersmann ist die lebende Vorlage für Waldorf oder Statler – er verrät aber nicht, für welchen der beiden. Er nickt aber auch schon mal bei einigen Showauftritten mit dem Kopf (sein Äquivalent zu »Applaus, Applaus, Applaus«), wenn bspw. Mr. John Allsopp sagt, dass *responsive design* und Webdesign doch dasselbe seien; oder wenn Mr. Jeremy Keith sagt, dass *progressive enhancement* ein wahrhaft befreiender Gedanke wäre. Gunnar lehnt sich auch gern mal beyond Logenrand … auf die Gefahr hin, abzustürzen und im Parkett aufzuschlagen.

@g16n |

HERON-VERFAHREN (BABYLONISCHES WURZELZIEHEN)

$$x_0 \approx \sqrt{a} > 0$$

$$x_{n+1} = (x_n + a \,/\, x_n) \,/\, 2$$

$$\lim_{n \to \infty} x_n = \sqrt{a}$$

```
@function sqrt($a)
{
  $x: 1;

  @for $i from 1 through 3
  {
    $x: ($x + $a / $x) / 2;
  }

  @return $x;
}
```

HERON-VERFAHREN (BABYLONISCHES WURZELZIEHEN)

$$x_0 \approx \sqrt{a} > 0$$

$$x_{n+1} = (x_n + a \,/\, x_n) \,/\, 2$$

$$\lim_{n \to \infty} x_n = \sqrt{a}$$

```
@function sqrt($a)
{
  $x: 1;


    $x: ($x + $a / $x) / 2;
    $x: ($x + $a / $x) / 2;
    $x: ($x + $a / $x) / 2;

  @return $x;
}
```

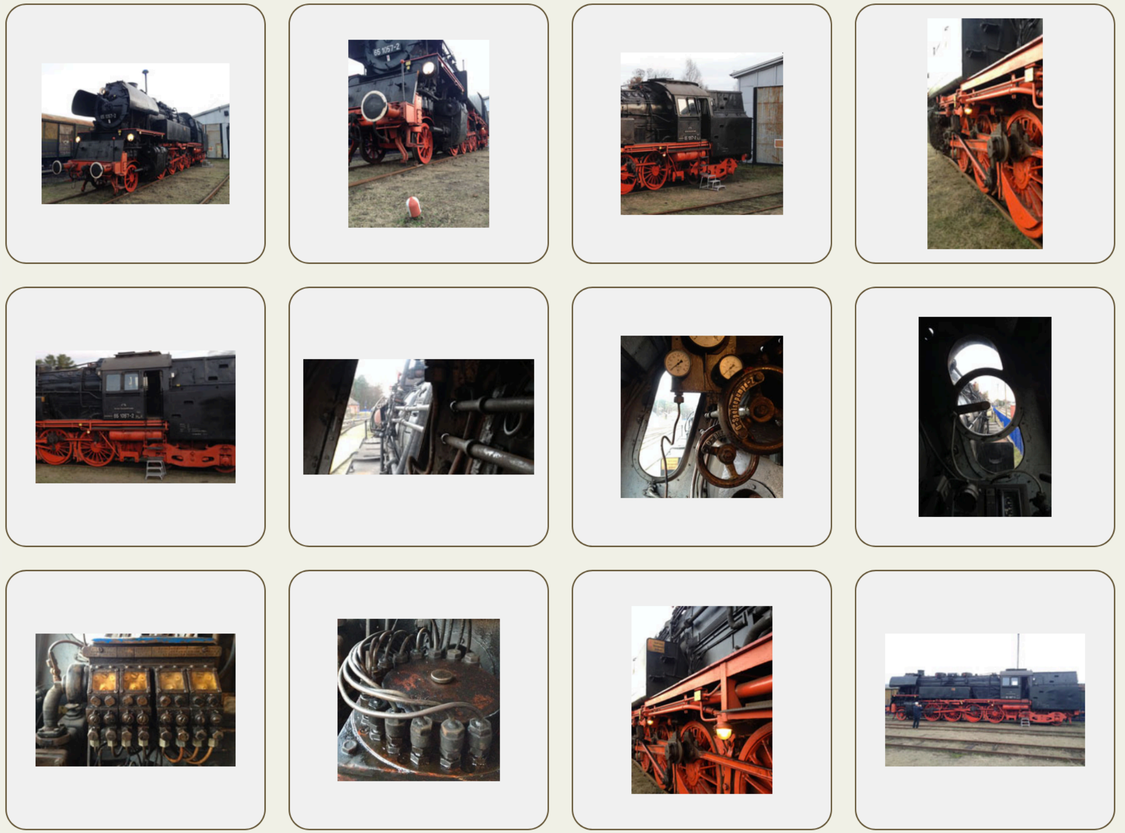
Heron-Verfahren (Babylonisches Wurzelziehen)

```
--area: 0.4;
--a: calc(var(--area) * var(--aspect-ratio));

--x0: 1;
--x1: calc((var(--x0) + var(--a) / var(--x0)) / 2);
--x2: calc((var(--x1) + var(--a) / var(--x1)) / 2);
--x3: calc((var(--x2) + var(--a) / var(--x2)) / 2);

width: calc(var(--x3) * 100%);
```

BEISPIEL: BILDERGALERIE



```
<img src="https://bittersmann.de/images/65.1057/580.jpg"
  style="--aspect-ratio: calc(4/3)" alt="" />
```

Das sind keine Inline-Styles.

```
<div class="container">
  <div class="row">
    <div class="col-12 col-md-8 col-lg-6">
```

Aber das sind Inline-Styles. 💩

WINKELFUNKTIONEN (TAYLOR-REIHEN)

$$\sin x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!} = + \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \pm \dots$$

$$\cos x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!} = + \frac{x^0}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} \pm \dots$$

$$\arctan x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{2k+1} = + \frac{x^1}{1} - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} \pm \dots$$

§ 11.4. Trigonometric Functions: ‘sin()’, ‘cos()’, ‘tan()’, ‘asin()’, ‘acos()’, ‘atan()’, and ‘atan2()’

The trigonometric functions—‘sin()’, ‘cos()’, ‘tan()’, ‘asin()’, ‘acos()’, ‘atan()’, and ‘atan2()’—compute the various basic trigonometric relationships.

The ***‘sin(A)’***, ***‘cos(A)’***, and ***‘tan(A)’*** functions all contain a single calculation which must resolve to either a <number> or an <angle>, and compute their corresponding function by interpreting the result of their argument as radians. (That is, ‘sin(45deg)’, ‘sin(.125turn)’, and ‘sin(3.14159 / 4)’ all represent the same value, approximately ‘.707’.) They all represent a <number>; ‘sin()’ and ‘cos()’ will always return a number between –1 and 1, while ‘tan()’ can return any number between –∞ and +∞. (See § 11.9 Type Checking for details on how math functions handle ∞.)

The ***‘asin(A)’***, ***‘acos(A)’***, and ***‘atan(A)’*** functions are the "arc" or "inverse" trigonometric functions, representing the inverse function to their corresponding "normal" trig functions. All of them contain a single calculation which must resolve to a <number>, and compute their corresponding function, interpreting their result as a number of radians, representing an <angle>. The angle returned by ‘asin()’ must be normalized to the range [‘-90deg’, ‘90deg’]; the angle returned by ‘acos()’ to the range [‘0deg’, ‘180deg’]; and the angle returned by ‘atan()’ to the range [‘-90deg’, ‘90deg’].

The ***‘atan2(A, B)’*** function contains two comma-separated calculations, A and B. A and B can resolve to any <number>, <dimension>, or <percentage>, but must have the *same* type, or else the function is invalid. The function returns the <angle> between the positive X-axis and the point (B,A). The returned angle must be normalized to the interval (‘-180deg’, ‘180deg’] (that is, greater than ‘-180deg’, and less than or equal to ‘180deg’).

> Note: ‘atan2(Y, X)’ is *generally* equivalent to ‘atan(Y / X)’, but it gives a better answer when the point in question may include negative components. ‘atan2(1, -1)’, corresponding to the point (-1, 1), returns ‘135deg’, distinct from ‘atan2(-1, 1)’, corresponding to the point (1, -1), which returns ‘-45deg’. In contrast, ‘atan(1 / -1)’ and ‘atan(-1 / 1)’ both return‘-45deg’, because the internal calculation resolves to ‘-1’ for both.

§ 11.5. Exponential Functions: ‘pow()’, ‘sqrt()’, ‘hypot()’, ‘log()’, ‘exp()’

The exponential functions—‘pow()’, ‘sqrt()’, ‘hypot()’, ‘log()’, and ‘exp()’—compute various exponential functions with their arguments.

The ***‘pow(A, B)’*** function contains two comma-separated calculations A and B, both of which must resolve to <number>s, and returns the result of raising A to the power of B, returning the value as a <number>.

The ***‘sqrt(A)’*** function contains a single calculation which must resolve to a <number>, and returns the square root of the value as a <number>. (‘sqrt(X)’ and ‘pow(X, .5)’ are basically equivalent, differing only in some error-handling; ‘sqrt()’ is a common enough function that it is provided as a convenience.)

The ***‘hypot(A, …)’*** function contains one or more comma-separated calculations, and returns the length of an N-dimensional vector with components equal to each of the calculations. (That is, the square root of the sum of the squares of its arguments.) The argument calculations can resolve to any <number>, <dimension>, or <percentage>, but must have the *same* type, or else the function is invalid; the result will have the same type as the arguments.

▸ **Why does ‘hypot()’ allow dimensions (values with units), but ‘pow()’ and ‘sqrt()’ only work on numbers?**

The ***‘log(A, B?)’*** function contains one or two calculations (representing the value to be logarithmed, and the base of the logarithm, defaulting to e), which must resolve to <number>s, and returns the logarithm base B of the value A, as a <number>.

The ***‘exp(A)’*** function contains one calculation which must resolve to a <number>, and returns the same value as ‘pow(e, A)’ as a <number>.

EXAMPLE 37
The ‘pow()’ function can be useful for strategies like CSS Modular Scale, which relates all the font-sizes on a page to each other by a fixed ratio.


BACK
to the
FUTURE